*// Last update: Finished chapter 23.*

**CHAPTER 10: ARRAYS**

Arrays are sequences of objects of the ***same type,*** *for example we can make an array that can store integers:*

*int main()  
{  
 int arr[10];  
}*

The above array can store 10 variables of integer type, and elements are adjacent to each other (or their addresses are adjacent to each other.)

To initialize an array, we can use the **initialization list {}**

*int main()  
{  
int arr[4] = {32, -12, 0, 4};  
}*

Elements are put in the curly brackets **{ },** and each elements are separated by the comma “ **,** ”.

The order of the elements is from the left to the right: 32 is the first element, -12 is the second element, 0 is the third element and 4 is the last element.

However, it should be noted that array indexing **starts from 0, not 1.** So when you want to use the first element of **arr**, type **arr[0].**

* If the length of the array is **N**, then the index starts **from 0 to N-1.**

We can insert elements to the array from user input by using loop such as **for loop**

*int main(){*

*int arr[10]; // Create an array of integers that has the length of 10*

*for(int i=0; i<10; ++i)*

*cin >> arr[i];*

*}*

**CHAPTER 11: POINTERS**

Every objects are store in memory, so each objects has its own **value** and **address**. When talking about how to store the **address** of that object, we use a special variable called **pointers.** So a pointer is a type that can hold the address of a particular object.

To create a pointer, we use the “\*” mark before the pointer’s name.

*int main()  
{  
int \*ptr;  
}*

Above is a pointer that can hold an **integer’s address**. Conversely, we can make a pointer that can hold **char’s address.**

*int main()  
{  
char \*ptr;  
}*

To make a pointer points to a specific variable, we use the “ **&** ” operator to get the variable’s address.

int main()  
*{  
int x = 123;  
int\* p = &x;  
}*

We can also initialize our pointer to let it points to **nowhere**, or **nullptr**.

*int main()  
{  
char\* p = nullptr;  
}*

p is now called a ***null pointer***.

To access the value of an object which is being pointed to by a pointer, we can use the “ **\*** ” operator.

*#include <iostream>*

*using namespace std;*

*int main(){*

*int a = 5;*

*int \*ptr = &a;*

*cout << “The value of a is:” << \*ptr; // Prints out 5*

*}*

We can also change the value of the object through pointer.

#include <iostream>

using namespace std;  
int main()  
{  
int x = 123;  
int\* p = &x;  
\*p = 456; // change the value of pointed-to object  
cout << "The value of x is: " << x; // Prints out 456  
}

**CHAPTER 12: REFERENCES**

A reference type is an alias to an existing object in memory, and reference must be initialized. For example:

*int main()  
{  
 int x = 65;  
 int& y = x;  
}*

Now we have two different names that refer to the same integer. If we assign or change value one of them, the value of both will be changed. In other words, they are using two different names for the same value.

*int main()  
{  
 int x = 123;  
 int& y = x;  
 x = 456;  
 // both x and y now hold the value of 456  
 y = 789;  
 // both x and y now hold the value of 789  
}*

Note: The “ **\*** ” operator is used when creating a pointer:

int \*ptr;

**But**, it is also used to refer to the value of the object being pointed by a pointer.

\*ptr = 350;

The “ **&** ” operator is used to get the address of an object;

ptr = &myVar;

**But**, it is also used as initializing a reference.

int &y = x;

* It is important to look at the **context** to see what role does these operators play.

**CHAPTER 13: INTRODUCTION TO STRINGS.**

C++ library offers a compound type called string or rather std::string as it is part of the std namespace. We use it for storing and manipulating string of texts.

**Defining a string:**

To use the std::string type, we need to include the <string> header in our program.

*#include <iostream>  
#include <string>  
int main()  
{  
std::string s = "Hello World.";  
std::cout << s;  
}*

**Concatenating strings:**

We can add a string literal to our string using the compound operator “**+=**”

*#include <iostream>  
#include <string>  
int main()  
{  
std::string s = "Hello ";  
s += "World.";  
std::cout << s;  
}*

We can also connect 2 (or more) strings together with “**+**”

*#include <iostream>  
#include <string>  
int main()  
{  
std::string s1 = "Hello ";  
std::string s2 = "World.";  
std::string s3 = s1 + s2;  
std::cout << s3; // Prints out Hello World.  
}*

**Accessing Characters:**

Individual characters of a string can be accessed through a **subscript operator []** or via a  
member function ***.at(index)*.** The index starts at 0. Example:*#include <iostream>  
#include <string>  
int main()  
{  
std::string s = "Hello World.";*

|  |  |
| --- | --- |
| *char c1 = s[0];  char c2 = s.at(0);  char c3 = s[6];  char c4 = s.at(6);* | *// 'H' // 'H'; // 'W' // 'W';* |

*std::cout << "First character: " << c1 << ", sixth character: " << c3;  
}*

**Comparing Strings:**

A string can be compared to string literals and other strings using the equality “**==**” operator. Comparing a string to a string literal: *#include <iostream>  
#include <string>  
int main()  
{  
 std::string s1 = "Hello";  
 if (s1 == "Hello")  
 {  
 std::cout << "The string is equal to \"Hello\"";  
 }  
}*

*Comparing a string to another string is done using the equality operator “****==****”  
#include <iostream>  
#include <string>  
int main()  
{  
 std::string s1 = "Hello";  
 std::string s2 = "World.";  
 if (s1 == s2)  
 {  
 std::cout << "The strings are equal.";  
 }  
 else  
 {  
 std::cout << "The strings are not equal.";  
 }  
}*

***String input:***

Preferred way of accepting a string from the standard input is via the *std::getline*  
function which takes std::cin and our string as parameters:*#include <iostream>  
#include <string>  
int main()  
{  
std::string s;  
std::cout << "Please enter a string: ";  
std::getline(std::cin, s);  
std::cout << "You entered: " << s;  
}*

We use the *std::getline* because our string can contain white spaces. And if we  
used the *std::cin* function alone, it would accept only a part of the string.

**A pointer to a string:**

A string has a member function .c\_str() which returns a pointer to its first element.

#include <iostream>  
#include <string>  
int main()  
{  
 std::string s = "Hello World.";  
 std::cout << s.c\_str();  
}

**Substrings:**

To create a substring from a string, we use the *.substr()* member function. The signature of the function is: *.substring(starting\_position, length) .*Example:

#include <iostream>  
#include <string>  
int main()  
{  
 std::string s = "Hello World.";  
 std::string mysubstring = s.substr(6, 5);  
 std::cout << "The substring value is: " << mysubstring; // Prints out World  
}

**Finding a substring:**

To find a substring in a string, we use the *.find()* member function. If the substring is found, the function returns the position of the first found substring. If the substring is not found, the function returns a value that is *std::string::npos*.

To find a substring “Hello” inside the “This is a Hello World string” string, we write:  
#include <iostream>  
#include <string>  
int main()  
{  
 std::string s = "This is a Hello World string.";  
 std::string stringtofind = "Hello";  
 std::string::size\_type found = s.find(stringtofind);  
 if (found != std::string::npos)  
 {  
 std::cout << "Substring found at position: " << found;  
 }  
 else  
 {  
 std::cout << "The substring is not found.";  
 }  
}

**CHAPTER 14: AUTOMATIC TYPE DEDUCTION**

We can automatically deduce the type of an object using the **auto** specifier. The **auto**specifier deduces the type of an object based on the object’s **initializer type.**Example:

|  |  |
| --- | --- |
| auto c = 'a'; | // char type |
| This example deduces c to be of type char as the initializer 'a' is of type char. Similarly, we can have: |  |
| auto x = 123; | // int type |

Here, the compiler deduces the x to be of type int because an integer literal 123 is of type int.  
The type can also be deduced based on the type of expression:  
auto d = 123.456 / 789.10; // double

This example deduces d to be of type double as the type of the entire 123.456 / 789.10 expression is double.  
We can use auto as part of the reference type:  
int main()  
{  
int x = 123;  
auto& y = x; // y is of int& type  
}

or as part of the constant type:  
int main()  
{  
const auto x = 123; // x is of const int type  
}

We use the *auto* specifier when the type (name) is hard to deduce manually or cumbersome to type due to the length.

**CHAPTER 15: EXERCISES**

**15.1: Array Definition:** Write a program that defines and initializes an array of five doubles. Change and print the values of the first and last array elements.

#include <iostream>  
int main()  
{  
double array[5] = { 3.14, -5.2, 0.05, 2.22, 50.0201 };  
arr[0] = 4.64;  
arr[4] = 2.17;  
std::cout << "The first array element is: " << arr[0] << '\n';  
std::cout << "The last array element is: " << arr[4] << '\n';  
}

**15.2:Pointer to an Object:** Write a program that defines an object of type double. Define a pointer that points to that object. Print the value of the pointed-to object by dereferencing a pointer.

#include <iostream>  
int main()  
{  
double myDouble = -5.12;  
double \*ptr = &myDouble;  
std::cout << "The value of the pointed-to object is: " << \*p;  
}

**15.3:Reference Type:** Write a program that defines an object of type double called mydouble. Define an object of reference type called myreference and initialize it with mydouble. Change the value of myreference. Print the object value using both the reference and the original variable. Change the value of mydouble. Print the value of both objects.

#include <iostream>  
int main()  
{  
double mydouble = 3.14;  
double& myreference = mydouble;  
myreference = 1.43;  
std::cout << "The values are: " << mydouble << " and " << myreference  
<< '\n';  
mydouble = 7.35;  
std::cout << "The values are: " << mydouble << " and " << myreference << '\n';  
}

**15.4: Strings:** Write a program that defines two strings. Join them together and assign the result to a third-string. Print out the value of the resulting string.

#include <iostream>  
#include <string>  
int main()  
{  
std::string str1 = "Miles";  
std::string str2 = " Prower";  
std::string str3 = str1 + str2;  
std::cout << "The resulting string after joining is: " << str3;  
}

**15.5: Strings from standard input:**

Write a program that accepts the first and the last name from the standard input using  
the std::getline function. Store the input in a single string called fullname. Print out  
the string.

#include <iostream>  
#include <string>  
int main()  
{  
std::string fullname;  
std::cout << "Please enter your first name and your last name: ";  
std::getline(std::cin, fullname);  
std::cout << "Your full name is: " << fullname;  
}

**15.6:Creating a substring:** Write a program that creates two substrings from the main string. The main string is made up of first and last names and is equal to “John Doe.” The first substring is the first name. The second substring is the last name. Print the main string and two substrings afterward.

#include <iostream>  
#include <string>  
int main()  
{  
std::string fullname = "John Doe";  
std::string firstname = fullname.substr(0, 4);  
std::string lastname = fullname.substr(5, 3);  
std::cout << "The full name is: " << fullname << '\n';  
std::cout << "The first name is: " << firstname << '\n';  
std::cout << "The last name is: " << lastname << '\n';  
}

**15.7:Finding a single character:**

Write a program that defines the main string with a value of “Hello C++ World.” And checks if a single character ‘C’ is found in the main string.

#include <iostream>  
#include <string>  
int main()  
{  
 std::string s = "Hello C++ World.";  
 char c = 'C';  
 auto characterIsFound = s.find(c);  
 if (characterIsFound != std::string::npos)  
 {  
 std::cout << "Character found at position: " << characterIsFound <<'\n';  
 }  
 else  
 {  
 std::cout << "Character was not found." << '\n';  
 }  
}

**15.8:Finding a substring:**

Write a program that defines the main string with a value of “Hello C++ World.” and  
checks if a substring “C++” is found in the main string.

#include <iostream>  
#include <string>  
int main()  
{  
 std::string s = "Hello C++ World.";  
 std::string mysubstring = "C++";  
 auto mysubstringfound = s.find(mysubstring);  
 if (mysubstringfound != std::string::npos)  
 {  
 std::cout << "Substring found at position: " << mysubstringfound <<'\n';  
 }  
 else  
 {  
 std::cout << "Substring was not found." << '\n';  
 }  
}

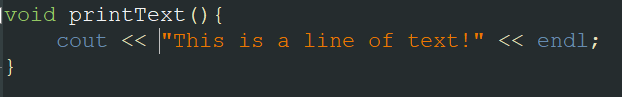
**15.9:Automatic Type Deduction:**

Write a program that automatically deduces the type for char, int, and double objects  
based on the initializer used. Print out the values afterward.#include <iostream>  
int main()  
{  
 auto c = 'a';  
 auto x = 123;  
 auto d = 3.14;  
std::cout << "The type of c is deduced as char, the value is: "<< c <<'\n';  
std::cout << "The type of x is deduced as int, the value is: "<< x <<'\n';  
std::cout << "The type of d is deduced as double, the value is: "<< d <<'\n';  
}

**CHAPTER 19: FUNCTIONS.**

When working on big projects, you will need many different functions to do many different things. At the start of this book till now, we have only coded in the **main**() function. In reality, we break down our code to many smaller modules, or smaller chunks called **functions**.

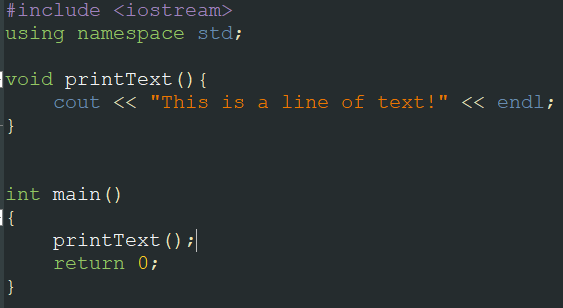
Here is an example of a function:



As the function name suggests, this function’s job is to print out text to the console.

This function has the type called **void**, which means you don’t have to return a value back to the function.

To use this function, we will need to call this printText() function in main().



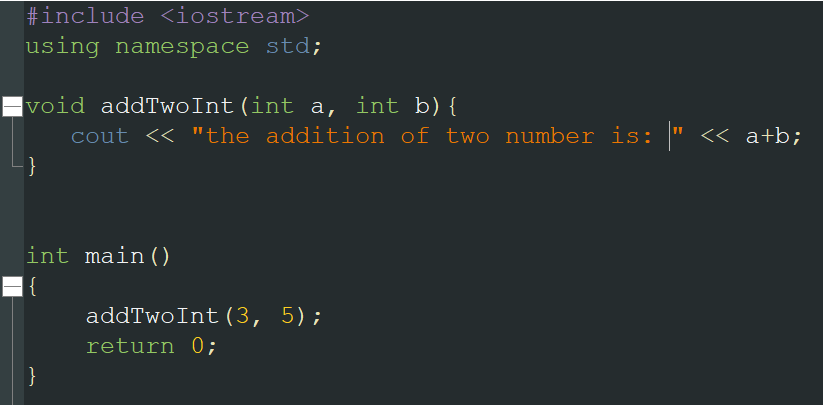
To call a function, type the function name follow with paranthesis.

The result is shown below:



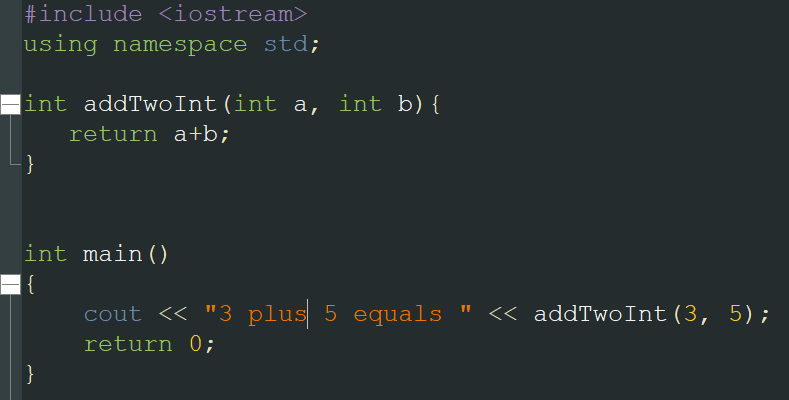
Usually, we would want our functions to calculate something, which means it will need values that are **passed** to them. These values are called **parameters**.

To specify types of parameters, we put them in the **parenthesis** of the function.



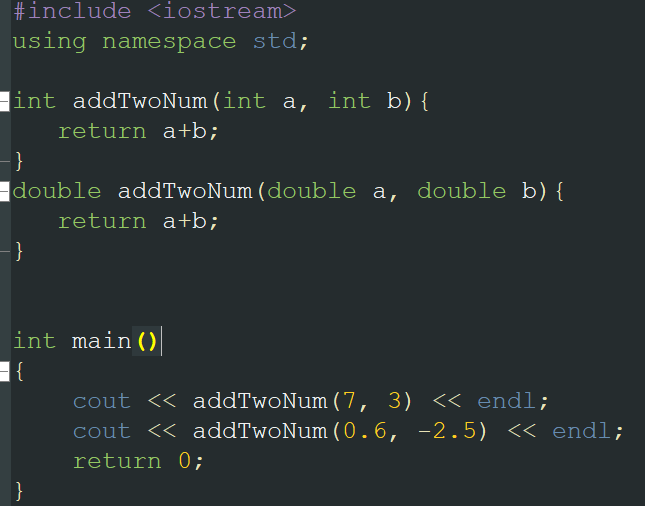


We can also use function type **int** to return the value:





We can also have many functions with the same name but with different parameters types, this is called **function overloading.** Below is an example of function overloading.



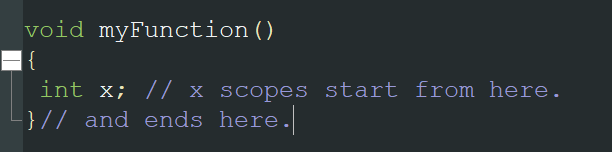


When passing different parameters, appropriate function is used accordingly.

**CHAPTER 21: SCOPE AND LIFETIME.**

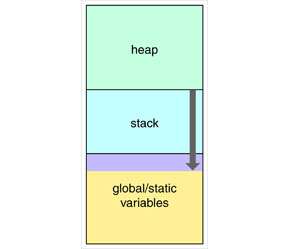
When declaring a variable, the variable can only be accessed inside it’s **scope.** There are many different scopes.

When declaring a variable inside a function, it has a **local scope.**



The **block-scope** is a section of code marked by a block of code starting with { and ending  
with }

When a object is stored in memory, its time which it lasts in the memory is called its **Lifetime (Storage Duration).**

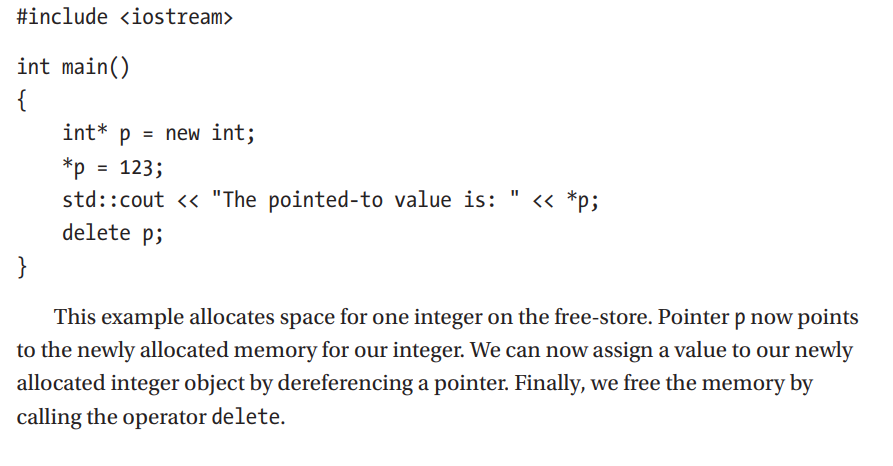


The automatic storage duration is a duration where memory for an object is  
automatically allocated at the beginning of a block and deallocated when the code  
block ends. This is also called a ***stack memory*;** objects are allocated on the *stack*. In this  
case, the object’s lifetime is determined by its scope. All local objects have this storage  
duration.

The dynamic storage duration is a duration where memory for an object is manually  
allocated and manually deallocated. This kind of storage is often referred to as ***heap  
memory*.** The user determines when the memory for an object will be allocated, and  
when it will be released. The lifetime of an object is not determined by a scope in which  
the object was defined. We do it through operator ***new* and *smart pointers***. In modern  
C++, we should prefer the smart pointer facilities to operator **new.**

When an object declaration is prepended with a **static** specifier, it means the storage  
for a static object is allocated when the program starts and deallocated when the  
program ends. There is only one instance of such objects, and (with a few exceptions)  
their lifetime ends when a program ends. They are objects we can access at any given  
time during the execution of a program.

We can dynamically allocate and deallocate storage for our object and have pointers  
point to this newly allocated memory.

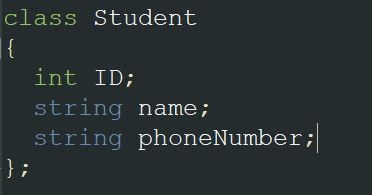


**CHAPTER 23: CLASSES – INTRODUCTION**

When creating many values for a certain type of objects, we usually have to **redundantly** declare the same type of value over and over again (For example, the IDs of students in a university.). We would want to make a **prototype**, or a **blueprint** for this object type so we don’t have to declare thousands of variables.

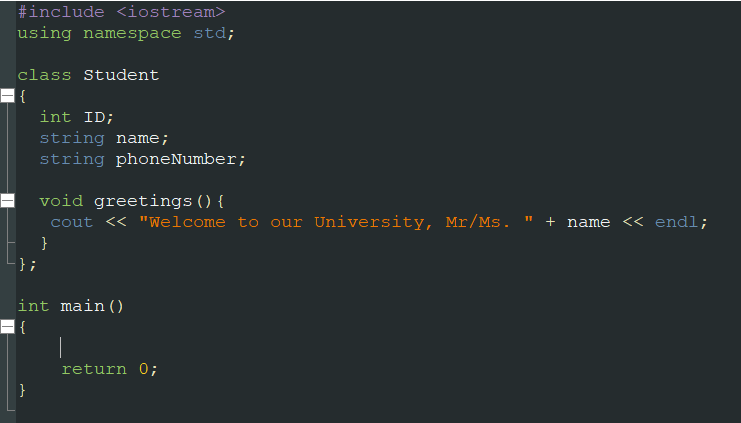
This is what we called a **class.**

**Let’s make a class that holds the information of a student.**



Variables that are inside a class are called **members** of the class.

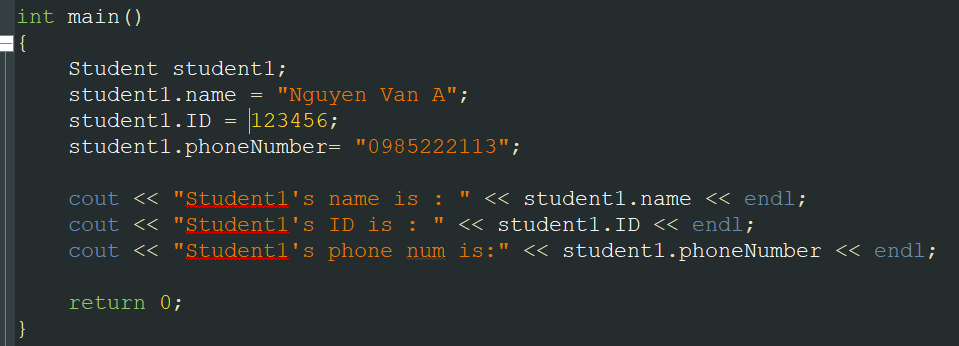
We can also put function inside a class, for example:

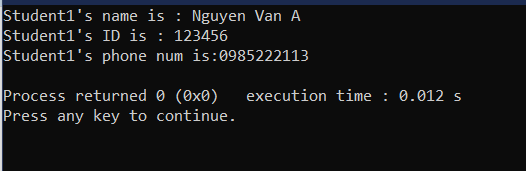


Now that we have our class, we would like to make an object (an instance of that class). To create an object named student1:



After creating an object, we can modify the value of that object as long as it is **public.**





We can modify the privilege to access the members of a class, that is called **access modifier.**

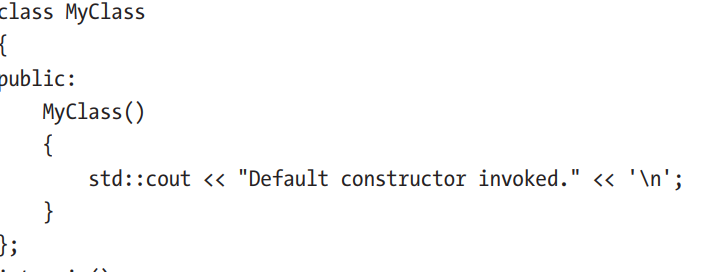
For now, there are two types of access modifier, **public and private.**

**Public** means members can be accessed anywhere **outside** of class (eg through an object).

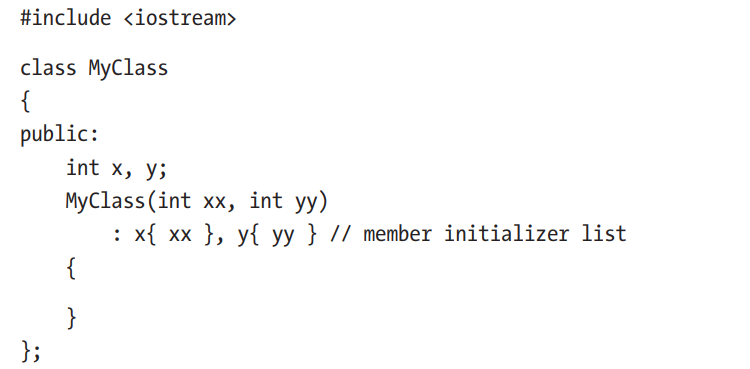
**Private** means members can only be accessed **inside** of its class.

A **constructor** is a member function that has the same name as the class. To initialize  
an object of a class, we use constructors.

By default, there is already a **default constructor**.

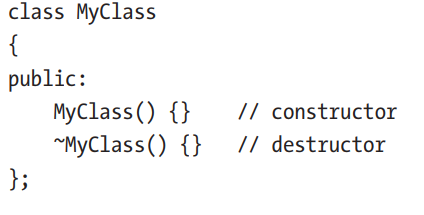


In our previous example, we used a constructor body and *assignments* to assign value to  
each class member. A better, more efficient way to initialize an object of a class is to use  
the constructor’s *member initializer list* in the definition of the constructor:



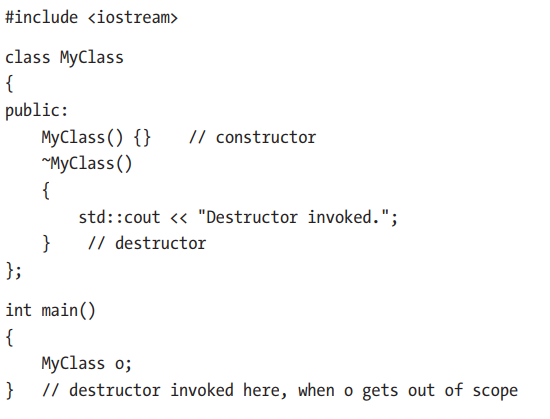
There are also many operations we can do with objects such as **copy,** **move and operator overloading.**

As we saw earlier, a constructor is a member function that gets invoked when the object  
is initialized. Similarly, a **destructor** is a member function that gets invoked when an  
object is **destroyed**.



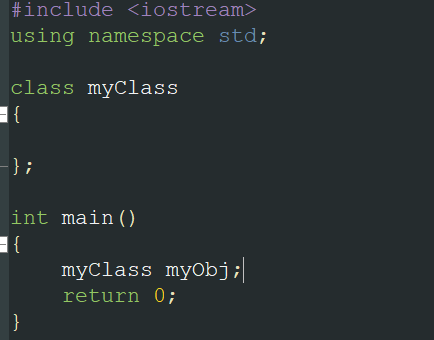
Destructor takes no parameters, and there is one destructor per class.

Destructors are called when an object goes **out of scope** or when a pointer to an  
object is deleted. We should not call the destructor directly.

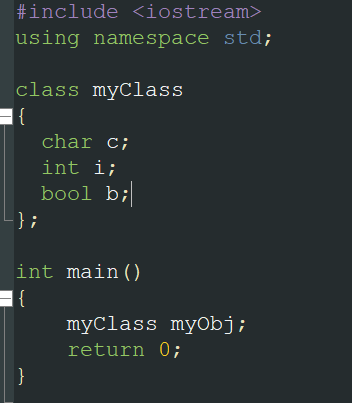


**CHAPTER 24: EXERCISES**

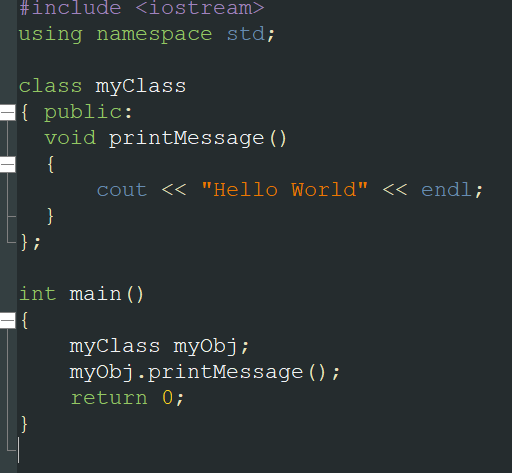
**24.1 Class Instance**



24.2 Class with data members



24.3 Class with member function

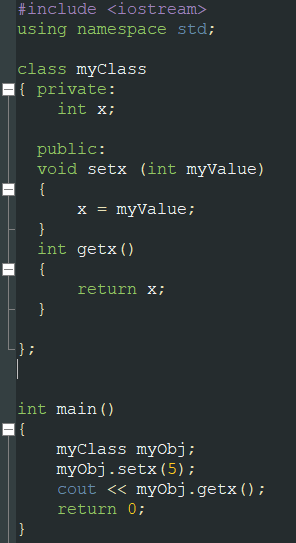


24.4 Class with data and Function members

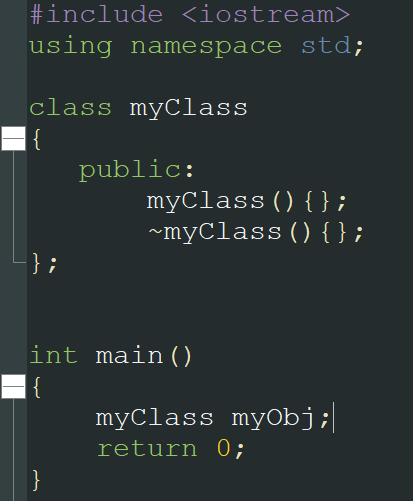




24.5 Class access modifier



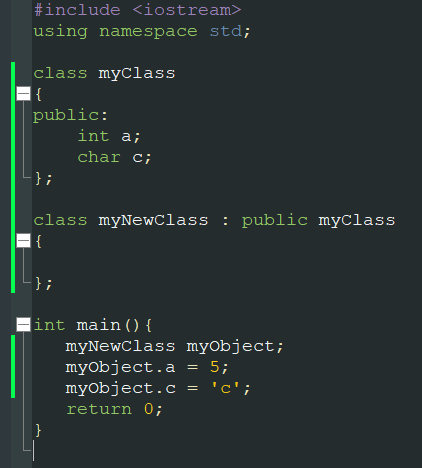
24.6 User-Defined Default Constructor and Destructor



Chapter 25: Classes: Inheritance and Polymorphism.

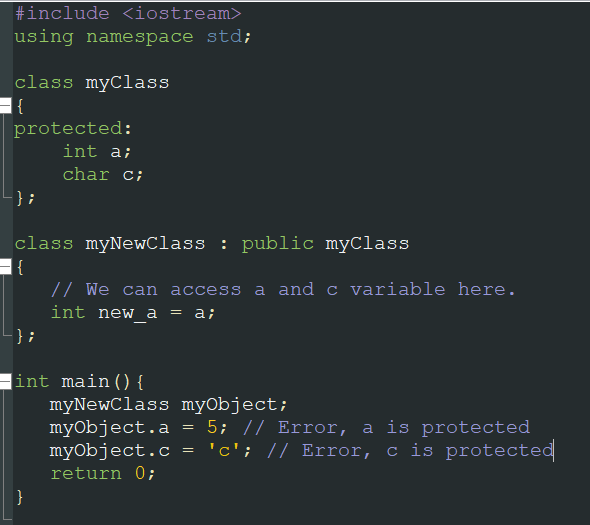
25.1 Inheritance:

Derived class and objects of a derived class can access public members of a base class:



In the above example, myNewClass is derived from myClass, and myNewClass can access int a and char c.

We now introduce a new access Specifier named Protected.

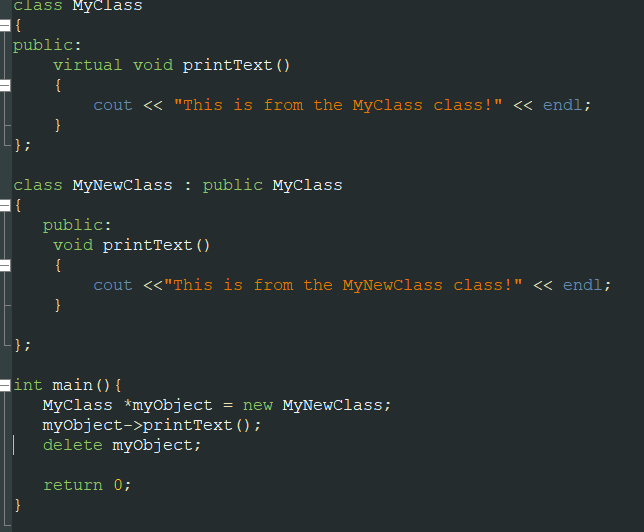


Protected fields cannot be accessed through objects, but can be accessed by derived class.

We can create new fields in the derived class, and those fields can only be accessed via that derived class, not the base class.

We can also derived a derived class.

25.2 Polymorphism



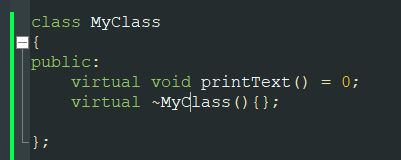


Virtual means this function can be overridden/redefined in subsequent derived classes, and the appropriate version will be invoked through a polymorphic object. In our main program, we create an instance of a MyNewClass class through a  
base class pointer. Using the arrow operator -> we invoke the appropriate version of the function. Here the myObject object morphs into different types to invoke the appropriate function. Here it invokes the derived version. That is why the concept is called polymorphism.

If there were no printText() function in the derived class, it would invoke the base class version.

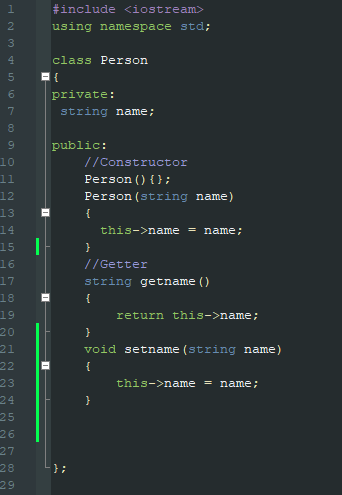
Functions can be *pure virtual* by specifying the = 0. Pure virtual functions must be re-defined in the derived class

One important thing to add is that a base class must have a virtual destructor if it is to be used in a polymorphic scenario. This ensures the proper deallocation of objects accessed through a base class pointer via the inheritance chain.

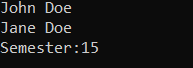


***Chapter 26: Exercises.***

Write a program that defines a base class called Person. The class has the following members:  
– A data member of type *std::string* called *name*– A single parameter, user-defined constructor which initializes the *name*– A getter function of type *std::string* called *getname(),* which returns the *name’s* value  
Then, write a class called *Student,* which inherits from the class *Person*. The class *Student* has the following members:  
– An integer data member called *semester*– A user-provided constructor that initializes the *name* and *semester* fields  
– A getter function of type *int* called *getsemester(),* which returns the *semester’s* value



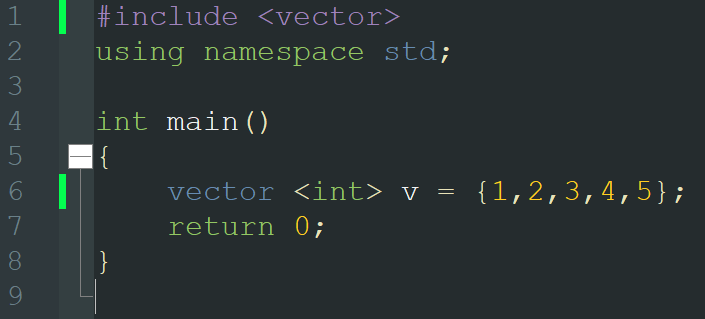




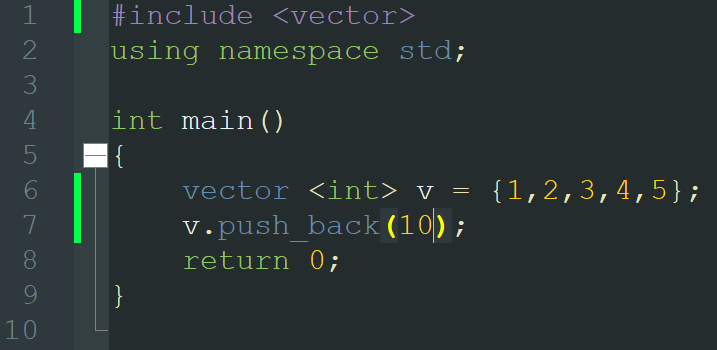
**38.1.1 std::vector**

Vector is a container defined in <vector> header. A vector is a sequence of contiguous elements.

A vector and all other containers are implemented as class templates allowing for storage of (almost) any type.



Vector can grow and shrink on its own as we insert and delete elements into and from a vector. To insert an element at the end of the vector, we use the vector's .push\_back() member function.



Vector’s size as a number of elements, can be obtained through a .size() member function:

